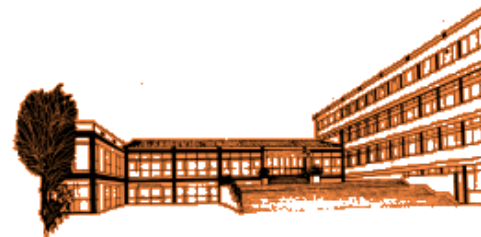




ISEL

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA



Instituto Superior de Engenharia de Lisboa



Infraestruturas Computacionais Distribuídas



Comunicação entre Processos - Sockets



Objetivos

- Transmitir conceitos importantes sobre comunicação entre processos através de uma rede de computadores.
- Apresentar a interface de programação com *sockets*, disponível no ambiente de execução Java.
- Focar a comunicação baseada em *sockets* suportados pelos protocolos da família TCP/IP.



Sumário

- ❏ Conceitos introdutórios: protocolos e endereçamento.
- ❏ Apresentação da interface de programação com *sockets*.
Implementação na linguagem Java.
- ❏ Aspectos a considerar na escolha do protocolo de comunicação, circuito virtual (TCP) ou datagrama (UDP).
- ❏ Exemplos de utilização de *sockets*, tendo como referência a arquitectura *cliente/servidor* e o ambiente de execução Java.



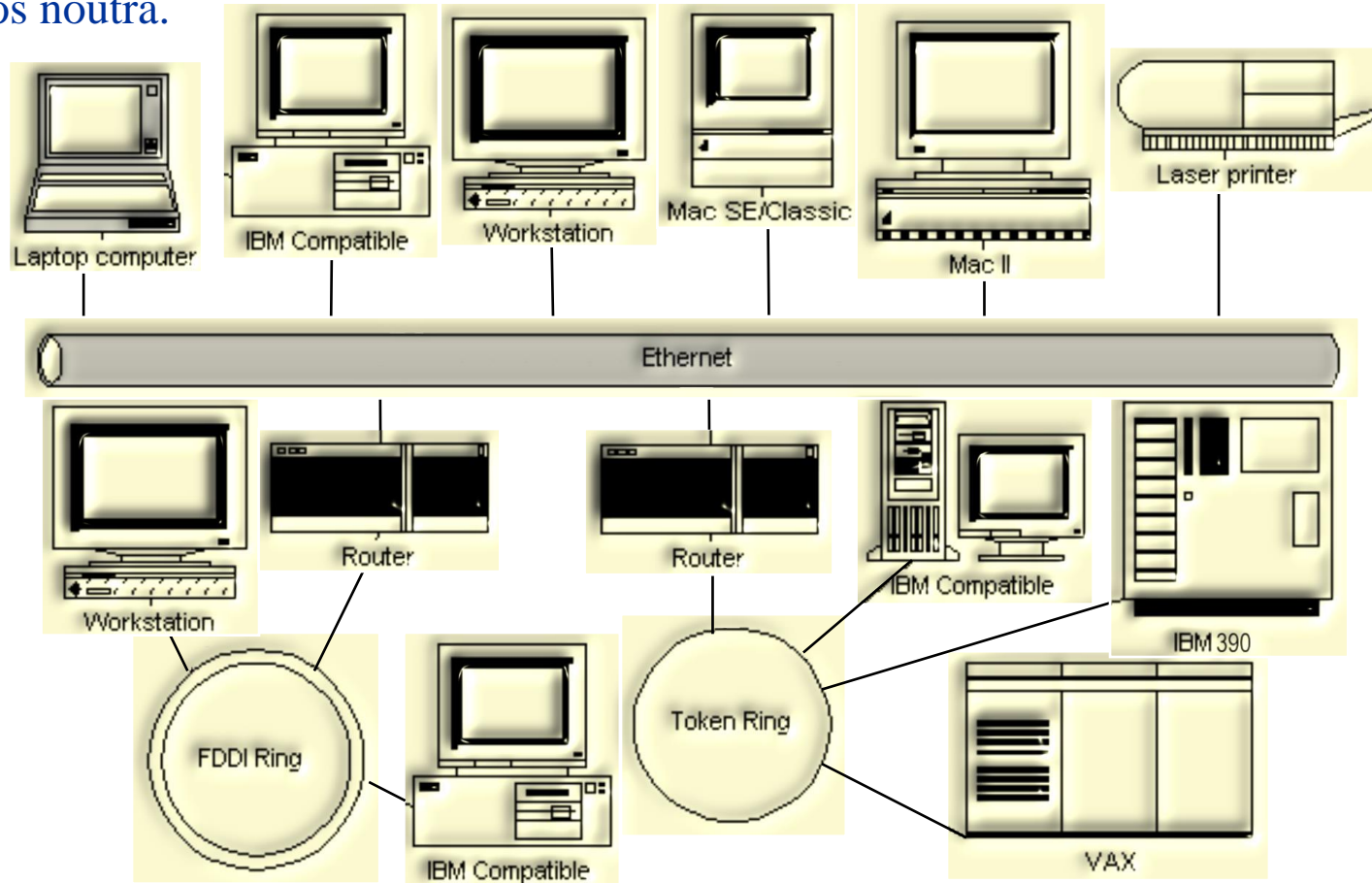
Redes

- Uma rede de computadores é um sistema de comunicação que liga computadores, designados vulgarmente por *end-systems* ou *hosts*.
- Tradicionalmente são considerados dois tipos de redes:
 - Locais (LAN *Local Area Network* - sala, edifício, campus,)
 - ⊗ de alguns metros a alguns Kms
 - ⊗ altas velocidades: de 1 a 1000 Mbps
 - ⊗ baixas taxas de erros
 - ⊗ baixo custo
 - ⊗ geralmente são privadas
 - De longa distância (WAN *Wide Area Networks* - país, continente, ...)
 - ⊗ de alguns Kms a milhares de Kms
 - ⊗ velocidades relativamente mais baixas
 - ⊗ taxas de erros mais significativas
 - ⊗ custo relativamente mais elevado
 - ⊗ geralmente são de utilização partilhada



Noção Genérica de Rede

- Uma *internet* ou *internetwork* é uma ligação entre duas ou mais redes distintas, que permite a comunicação entre computadores ligados numa rede com computadores ligados noutra.



Existe heterogeneidade na arquitectura das várias máquinas!



Heterogeneidade

- ❑ Não existe a garantia que os tipos de dados têm a mesma dimensão. Exemplo:
 - ❑ A dimensão dos tipos **short**, **int** e **long** diferem caso a máquina seja a *32-bit* ou a *64-bit*.
- ❑ Diferentes formas de empacotar as estruturas dependendo do número de *bits* de cada tipo de dados e das restrições de alinhamento de cada máquina.

Devido aos problemas anteriormente expostos não é aconselhável enviar estruturas binárias através de uma rede.



Heterogeneidade

- Existem duas soluções vulgarmente adoptadas para contornar os problemas anteriormente expostos:
 - Converter toda a informação numérica para caracteres. Assumindo que no mínimo as máquinas envolvidas na comunicação usam o mesmo conjunto de caracteres.
 - ⊗ Exemplo: ASCII ou Unicode (UTF-?).
 - ⊗ A forma comum de garantir interoperabilidade entre sistemas é de transferir documentos XML mais recentemente JSON.
 - Definir explicitamente o formato binário dos tipos de dados manipulados (número de *bits e Big ou Little Endian*) e converter os dados envolvidos na comunicação para este formato.
- A chamada de procedimentos remotos (***RPC Remote Procedure Call***) também usa a técnica de estabelecer o formato binário dos tipos de dados, mas de forma normalizada (***XDR External Data Representation***).



Heterogeneidade

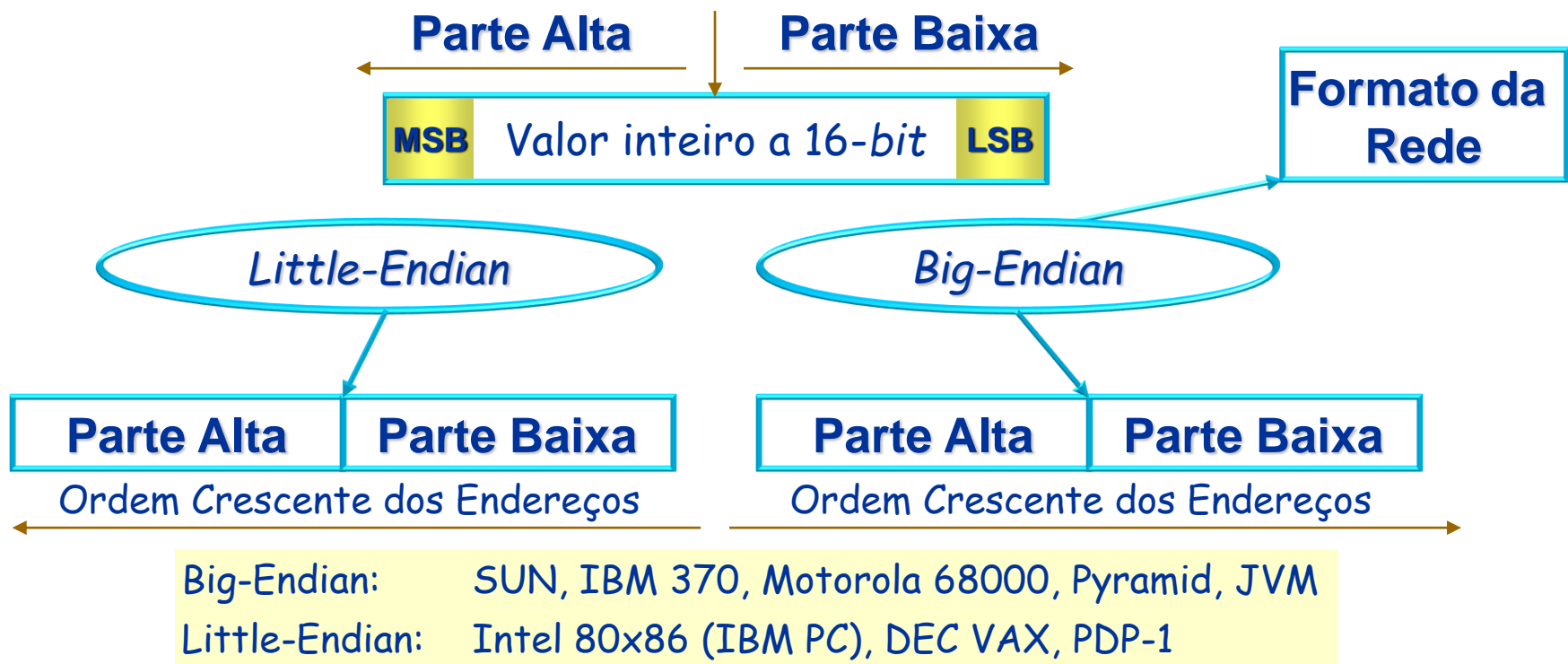


A heterogeneidade da arquitectura das várias máquinas ligadas em rede levanta alguns problemas:

- Podem existir diferentes formas de representar números (inteiros ou reais) em binário.

Exemplo:

- O formato de inteiros mais vulgar é o *Little-Endian* e o *Big-Endian*.





Ordem dos bytes da arquitectura de suporte nativa

Exemplo da representação em “Little-Endian”:

```
char  c1 = 1;  
char  c2 = 2;  
short s = 255; // 0x00FF  
long  l = 0x44332211;
```

Offset :	Memory dump
0x0000 :	01 02 FF 00
0x0004 :	11 22 33 44

Podemos descobrir qual a ordem dos bytes da arquitectura nativa através da classe `java.nio.ByteOrder`

```
import java.nio.ByteOrder;  
  
public class FindByteOrder {  
    public static void main(String[] args) {  
        ByteOrder bo = ByteOrder.nativeOrder();  
        if( bo.equals(ByteOrder.LITTLE_ENDIAN)) {  
            System.out.println(" Little Endian (intel)");  
        }else{  
            System.out.println(" Big Endian  
(Rede/Sparc/MVJ/Motorola)");  
        }  
    }  
}
```



Famílias de Protocolos

- ❏ Os computadores ligados em rede usam protocolos para comunicarem. Um protocolo é um conjunto de regras e convenções às quais os intervenientes na comunicação devem obedecer, quando trocarem informação.

- ❏ A definição de um protocolo envolve a definição de:
 - Ⓢ Sintaxe (formatos das mensagens)
 - Ⓢ Semântica (controlo, significados, acções, ...)
 - Ⓢ Temporizações

- ❏ Os protocolos são agrupados por famílias, exemplos:
 - Ⓢ TCP/IP (protocolos da Internet DARPA)
 - Ⓢ *Xerox Network Systems* (Xerox NS or XNS)
 - Ⓢ *IBM's System Network Architecture* (SNA)
 - Ⓢ *IBM's NetBios*
 - Ⓢ Protocolos OSI
 - Ⓢ *NetWare* IPX/SPX






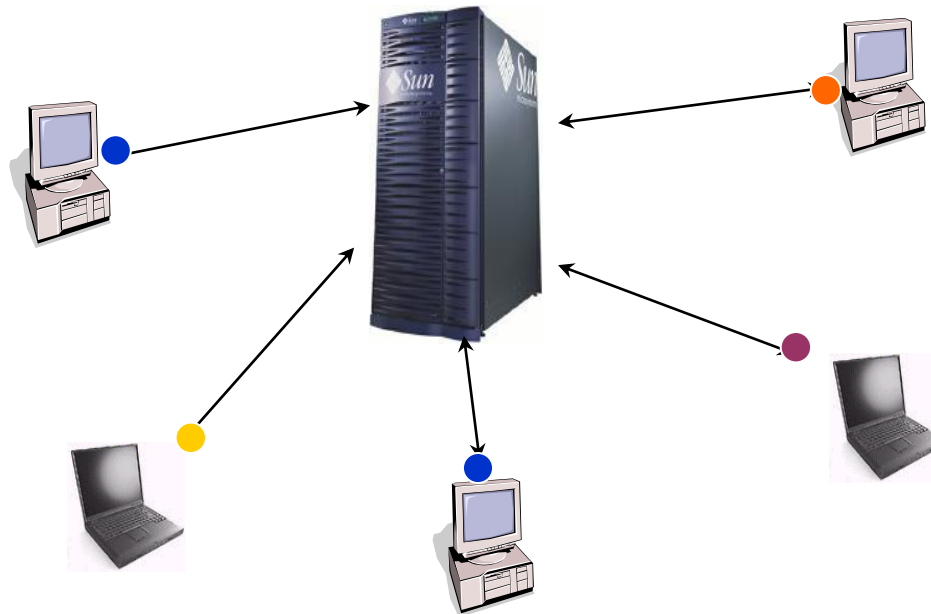
Protocolos de Transporte

- O nível de transporte é geralmente visível pelas aplicações pelo que é central na arquitectura de protocolos.
- Tal como o nível rede pode providenciar dois tipos de serviços
 - datagramas “*connection-less*”
 - circuito virtual “*connection oriented*”
- Vantagens
 - endereçamento “*end-to-end*”
 - oferecer uma interface de programação adequada às aplicações
 - melhorar a qualidade de serviço oferecida pelo nível rede e mascarar as suas deficiências (erros, quebras, etc.)



Arquitetura Cliente/Servidor

-  A arquitectura cliente/servidor é típica de aplicações concebidas para serem instanciadas num ambiente de rede.
-  O servidor é um processo que espera ser contactado pelo processo(s) cliente e tem como objectivo responder aos seus pedidos.
-  Numa arquitectura cliente/servidor, baseada em TCP/IP só o porto do servidor é importante ao nível aplicacional, o cliente ao ligar-se deve fornecer o endereço completo constituído pelo endereço de rede e número de porto.





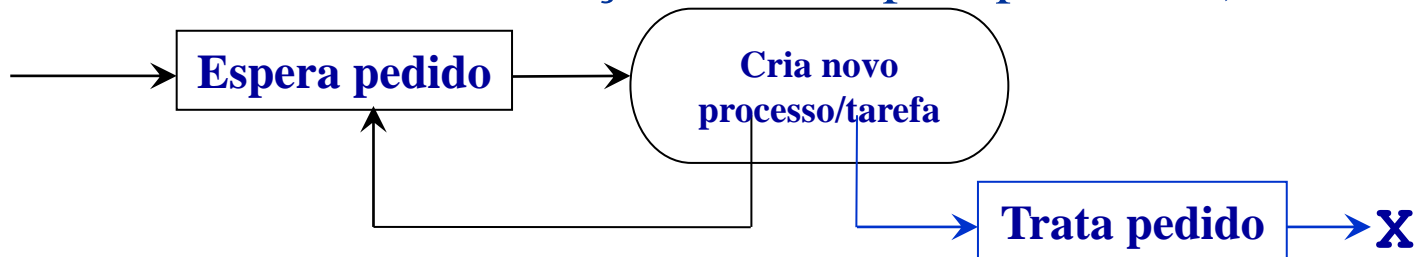
Servidor Iterativo Vs. Concorrente

Os servidores podem dividir-se em de 2 tipos:

Iterativo: quando o processamento do pedido demora pouco tempo, existindo um único processo/tarefa servidor para responder ao pedido



Concorrente: quando não se consegue prever o tempo de processamento do pedido, é criado um novo processo/tarefa para responder ao pedido (é necessário que o sistema operativo suporte simultaneamente a execução de múltiplos processos)





Thin client Vs Fat client

■ Numa arquitectura Cliente/Servidor podemos identificar dois tipos de abordagem na construção da aplicação cliente.

■ **Thin client** – A aplicação cliente apenas é responsável pela camada de apresentação da aplicação. Toda a lógica de negócio do sistema está centralizada no servidor.









- ⌚ Maior controlo e processamento dos dados no lado do servidor.
- ⌚ Os clientes podem ter poucos recursos computacionais ou não ser de confiança.

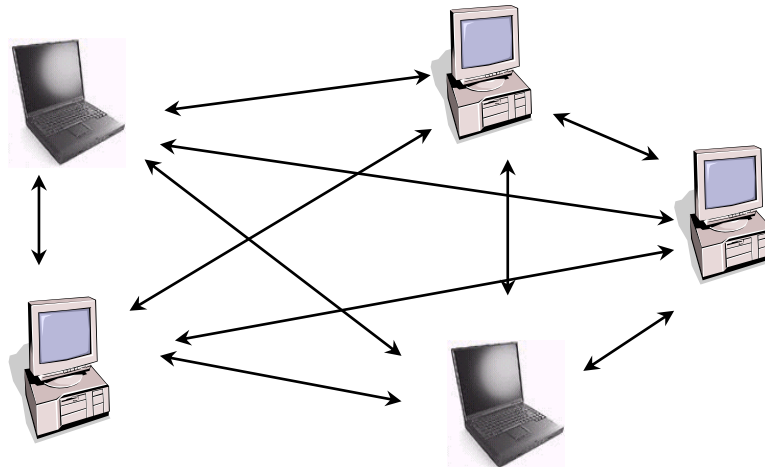
■ **Fat (or thick) client** – A aplicação cliente efectua cálculos correspondentes à lógica de negócio do sistema.

- ⌚ O servidor distribui a carga de trabalho pelas aplicações clientes, evitando ser o factor de estrangulamento do sistema. (poderá ser um factor a ter em conta quando for previsível o sistema estar em carga elevada)
- ⌚ As aplicações clientes têm de se executar em máquinas com recursos suficientes para poder suportar as operações pretendidas.



Arquitetura p2p: peer-to-peer

-  Todos os nós da rede são iguais entre si podendo requisitar ou fornecer serviços através da rede uns aos outros.
-  Não existe um nó central (vulnerável de falha) que seja responsável pela gestão da rede.
 -  A gestão terá de ser distribuída por toda a rede
-  Dificuldades deste tipo de arquitectura:
 -  Manutenção e descoberta dos endereços dos nós da rede;
 -  Manutenção de um estado global do sistema por toda a rede.
 -  Problemas de sincronismo
 -  Problemas de partições na rede





Endereços

- 🖥 Um computador é identificado univocamente na rede por endereços ou nomes (mais confortáveis para os utilizadores).
- 🖥 Os endereços dividem-se em:
 - 📁 Endereços físicos (*Physical ou hardware addresses*)
 - ⌚ caracterizam a interface física
 - ⌚ existem num espaço sem estruturação
 - ⌚ muitas vezes apenas têm significado contextual
 - ⌚ Ex: Mac address na Ethernet, BT device address (6 Bytes)
 - 📁 Endereços de rede (*Network layer ou logical addresses*)
 - ⌚ caracterizam um computador na rede
 - ⌚ geralmente são hierárquicos
 - ⌚ geralmente são dependentes do tipo de rede (endereços IP, endereços AppleTalk, ...)



Referência ao endereço IPv4

Os endereços de rede podem ter comprimento fixo ou comprimento variável, geralmente são hierárquicos.

Exemplos:

Endereços IP (comprimento fixo)

Inteiro a 32 bits: < Net ID >< Host ID >

Um endereço IP é habitualmente representado numa notação em que os 4 *bytes* (32 bits) aparecem separados por um ponto, designada por “*dot notation*” (representação ASCII).

Exemplo:

0x0102FF04 é escrito 1.2.255.4

Endereço de loopback (localhost)

127.0.0.1



Referência ao endereço IPv6

- Um endereço IP é habitualmente representado numa notação em que temos 8 grupos de 4 dígitos hexadecimais (128 bits / 16 Bytes) aparecem separados por dois pontos “:”
- 2^{128} (cerca de 3.4×10^{38} endereços)
- (representação ASCII).

Exemplo:

2001:0db8:85a3:0000:0000:8a2e:0370:7334

2001:db8:85a3:0:0:8a2e:370:7334

2001:db8:85a3::8a2e:370:7334

Endereço de loopback (localhost)

0000:0000:0000:0000:0000:0000:0000:0001

::1

Qualquer sequência de grupos de zeros pode ser substituída por “::”
A substituição só pode ocorrer uma vez por endereço.



Endereço IP em Java

Na API do Java a classe **java.net.InetAddress** representa um endereço IP, e permite-nos por exemplo:

- Ⓜ Obter o nome da máquina: `getHostName()`
- Ⓜ Obter o endereço em bruto (32bits ↔ byte[4]): `getAddress()`
- Ⓜ Resolver nomes e IP's
- Ⓜ Etc ...

Exemplo

```
import java.net.*;

public class NameResolver {
    public static void main(String[] args) {
        try{
            String hostname = "localhost";

            /* Nome -> InetAddress */
            InetAddress addr = InetAddress.getByName(hostname);
            String ip = addr.getHostAddress();
            System.out.println("O host: " + hostname + " tem o IP: " + ip);

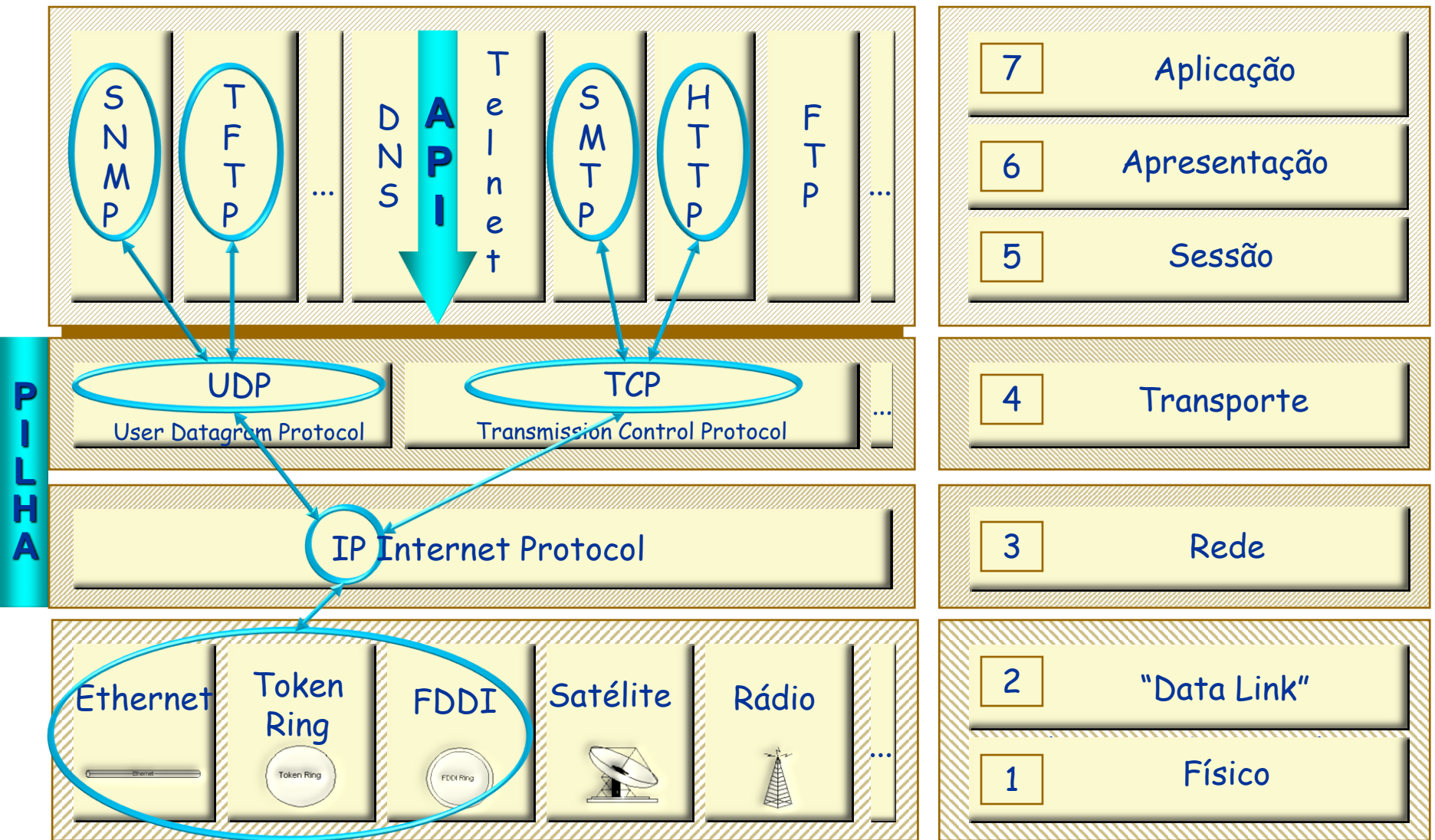
            /* InetAddress -> IP */
            byte[] rawIp = addr.getAddress();
            InetAddress addr2 = InetAddress.getByAddress(rawIp);
            System.out.println("O IP: " + ip + " tem o nome: " + addr2.getHostName());
        }catch(UnknownHostException uhe){
            System.err.println(uhe);
        }
    }
}
```

IP 32bits
ou
128bits

Aceita o nome e o IP
em “dot notation”



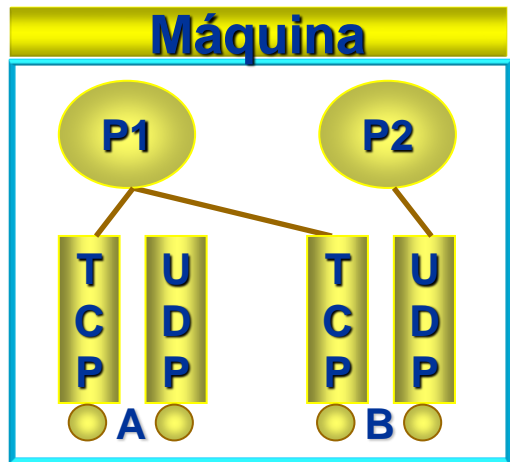
Protocolos da Família TCP/IP *versus* OSI





Noção de Porto

- Um endereço de rede identifica uma máquina, no entanto uma máquina pode ter mais do que um processo cada um com o seu número de porto (ou porta).



- Os portos em UDP e TCP correspondem a filas de mensagens independentes.

- Atribuição dos números dos portos nos sockets BSD:



- Existe outra distribuição de portos gerida pela IANA – *Internet Assigned Numbers Authority*:





Sockets

- ❏ O *socket* é um mecanismo de comunicação bidireccional com origem no UNIX *Berkley* 4.3BSD que permite a comunicação entre processos na mesma máquina ou entre máquinas distintas.
- ❏ A utilização de um *socket* em termos aplicativos é feita através de um descritor ou “*handle*”, da mesma forma que num ficheiro. Historicamente, os sockets são a extensão de uma das ideias mais importantes do Unix: que todas as E/S devem ser semelhantes às E/S nos ficheiros.
- ❏ Quando está ligado é caracterizado por:
{protocolo, endereço-local, porto-local, endereço-remoto, porto-remoto}
- ❏ Para iniciar a sua utilização cada processo deve criar a sua parte da infraestrutura *socket* que consiste em:
{protocolo, endereço-local, porto-local}
- ❏ Na comunicação são especificados 3 níveis de endereçamento:

socket = Endereço IP + Porto = Net ID + Host ID + Porto



Sockets em Java

 Podem realizar as seguintes operações:

 Abrir uma ligação (connect)

 Enviar dados (send)

 Receber dados (recv)

 Fechar uma ligação (close)

 Associar a um porto (bind)

 Esperar por dados

 Aceitar ligações de máquinas remotas (accept)

Clientes

Servidores

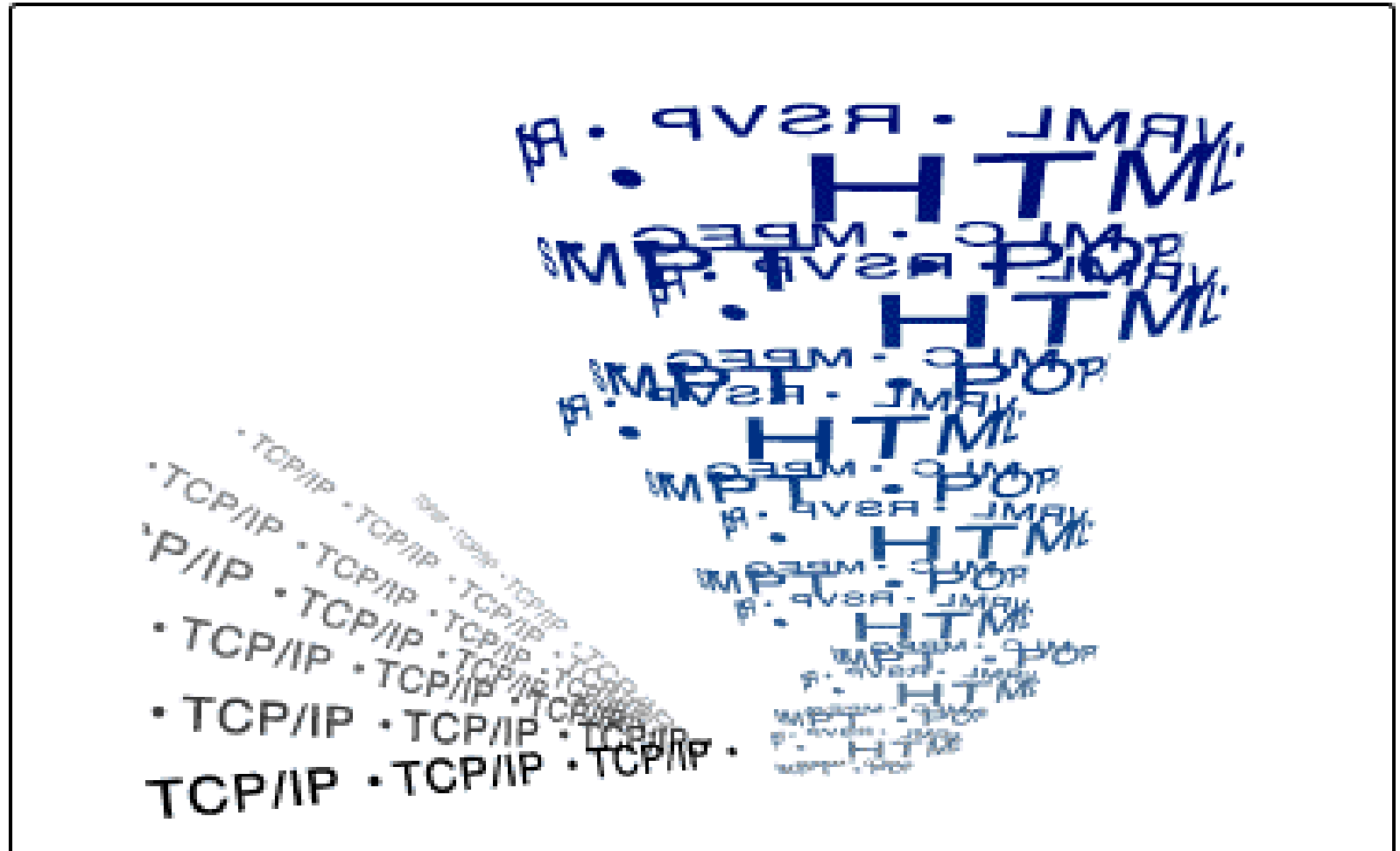
 Para efectuar ligações é necessário especificar além do endereço IP da máquina (*InetAddress*) o porto

Endereço

`{
InetAddress endereçoIp
int porto
}`



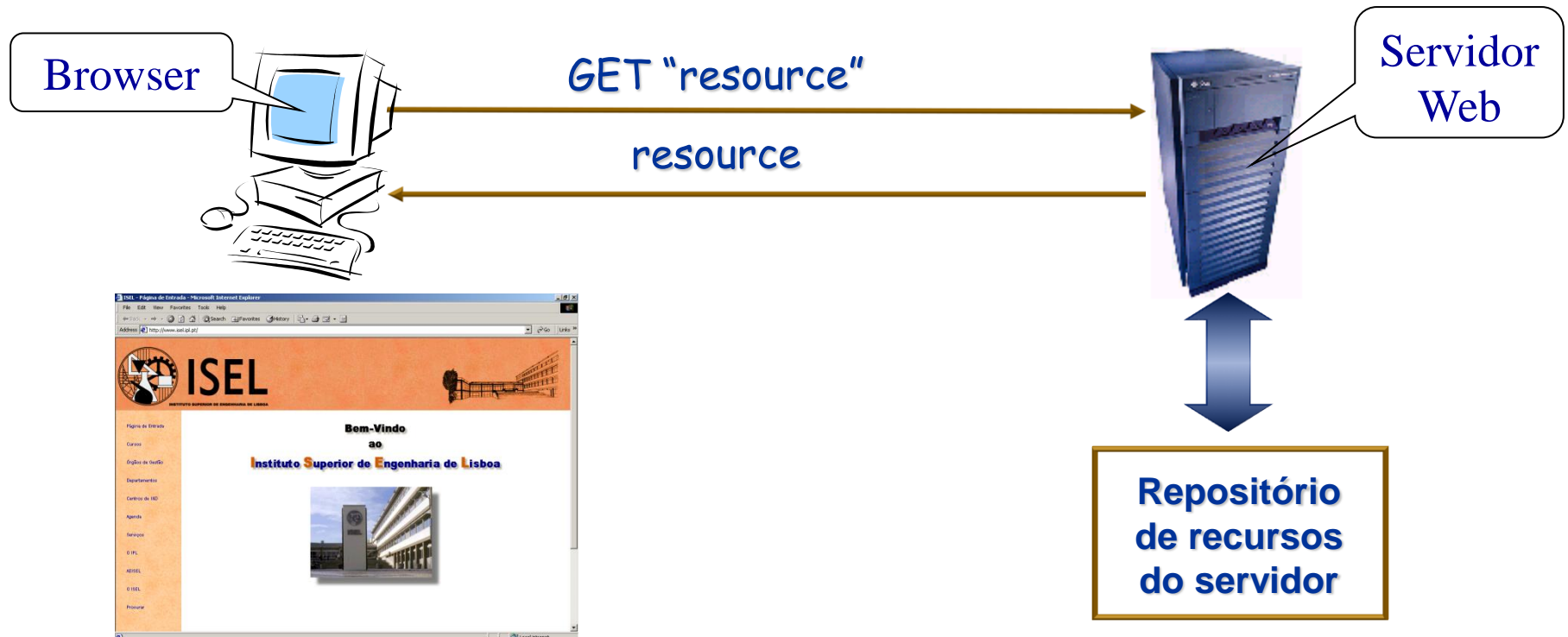
IP Tornado





Acesso a um Servidor HTTP

- O HTTP é um protocolo do tipo “request/reply” sobre uma conexão TCP. Tal como os outros protocolos da Internet é baseado em comandos textuais.



- O browser após receber o recurso (geralmente um documento ou ficheiro gif, jpeg, mpeg, etc...) interpreta-o e procede à sua visualização.



Características do Protocolo TCP

- 🖥️ Orientado à conexão (*connection oriented*), circuito virtual
- 🖥️ Não preserva a fronteira das mensagens (*stream-oriented*)
- 🖥️ Fiável (*reliable*)
 - 🖥️ garante a entrega
 - 🖥️ preserva a ordem de transmissão
 - 🖥️ não admite duplicados
- 🖥️ Bidireccional (*full-duplex*)
- 🖥️ Um processo pode trabalhar simultaneamente com várias conexões
- 🖥️ A terminação da conexão espera até todos os dados serem transmitidos

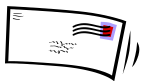


“Semelhante a realizar uma chamada telefónica”



Características do Protocolo UDP

- ❏ Ligação não permanente, não orientado à conexão (*connectionless*)
- ❏ Comunicação baseada em datagramas
- ❏ Preserva a fronteira das mensagens (*block-oriented*)
- ❏ Não fiável
 - ❏ não garante a entrega
 - ❏ não preserva a ordem de transmissão
 - ❏ podem ser recebidos duplicados
- ❏ Permite difusão genérica (*Broadcast*) e selectiva por grupos (*Multicast*)
- ❏ Em certos tipos de comunicação é mais eficiente que o TCP



“Semelhante a colocar cartas no correio, com remetente e destinatário”





SOCKETS

Application programmable interface (API)
em JAVA



Sockets em Java – Introdução

- O Java é uma Linguagem orientada a objectos e como tal temos classes que nos representam os Sockets
- O package *java.net* fornece uma framework OO para a criação e uso para os Sockets IP.
- Classes relacionadas com os sockets (java.net.*):
 - TCP:
 - ⊗ ServerSocket
 - ⊗ Socket
 - UDP:
 - ⊗ DatagramSocket
 - ⊗ DatagramPacket
 - ⊗ MulticastSocket
 - Endereçamento:
 - ⊗ InetAddress



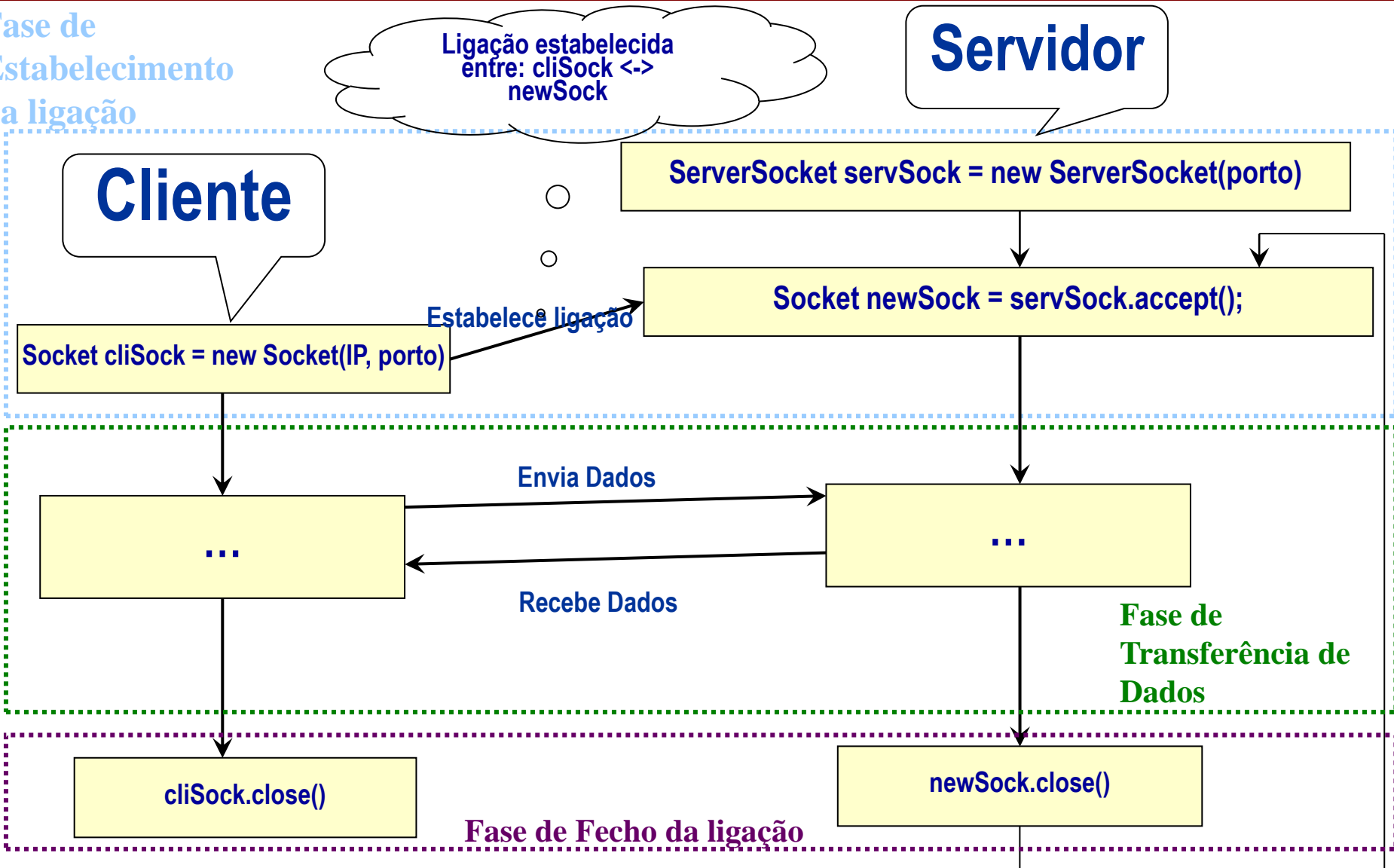
Sockets TCP (1)

- ❏ É diferenciado o socket cliente (que efectua a ligação) e o socket servidor (que espera ligações)
- ❏ O socket cliente é criado instanciando um objecto da classe “Socket” fornecendo no construtor ou o InetAddress ou o hostName e o porto, estabelecendo de imediato a ligação para o socket servidor
- ❏ O socket servidor é criado instanciando um objecto da classe “ServerSocket” fornecendo no construtor o porto onde vai esperar ligações. Contudo este socket só serve para estabelecer ligações dos clientes ao servidor. Após a aceitação de uma ligação com um cliente, esta passa a ser representada por um novo objecto da classe “Socket”. Este objecto é usado para a comunicação entre servidor e o cliente.



Sockets TCP (2)

Fase de Estabelecimento da ligação





Sockets TCP (3)

- Estabelecimento da ligação é efectuado no construtor do Socket.
Construtores mais utilizados:

- `public Socket(InetAddress address, int port) throws IOException`
- `public Socket(String host, int port) throws UnknownHostException, IOException`
- Etc...

- A transferência de dados é efectuada obtendo as streams de leitura e escrita no socket

- ## Leitura

- `@ public InputStream getInputStream() throws IOException`

- ## Escrita

- `@ public OutputStream getOutputStream() throws IOException`

- Fecho da ligação é feito recorrendo ao método close:

- `public void close() throws IOException`



Exemplo – cliente TCP (3)

```
import java.io.*;
import java.net.*;
```

```
public class Client_tcp {
    public static void main(String[] args) throws IOException {
        Socket sockfd = null;
        PrintWriter os = null;
        BufferedReader is = null;
        String host="localhost";
        int port=5025; // porto para ligar ao socket servidor

        try {
            sockfd = new Socket(host,port);
            // Mostrar os parametros da ligação
            System.out.println("Endereço do Servidor: " + sockfd.getInetAddress() + " Porto: " + sockfd.getPort());
            System.out.println("Endereço Local: " + sockfd.getLocalAddress() + " Porto: " + sockfd.getLocalPort());
            os = new PrintWriter(sockfd.getOutputStream(), true); // para escrita no socket
            is = new BufferedReader(new InputStreamReader(sockfd.getInputStream())); // para leitura do socket

        } catch (Exception e) {
            e.printStackTrace();
            System.err.println("Erro no estabelecimento da ligação:"+e.getMessage());
            System.exit(1);
        }
        os.println("Olá mundo!!!"); // Escreve no socket
        System.out.println("Recebi -> " + is.readLine()); // Mostrar o que se recebe do socket

        os.close();
        is.close();
        sockfd.close(); // No fim de tudo, fechar o socket
    }
}
```

**Criar Streams
para ler e
escrever do
socket**

**Cria socket
cliente e efectua
a ligação**

**Envia
mensagem
para socket**

**Recebe
mensagem do
socket**



Exemplo – servidor TCP (4)

```
import java.net.*;  
import java.io.*;
```

```
public class Server_tcp {  
    public static void main(String[] args) {  
        int port=5025;  
        PrintWriter os = null;  
        BufferedReader is = null;  
  
        try {  
            ServerSocket serverSocket = new ServerSocket(port);  
            for( ; ; ) {  
                System.out.println("Servidor aguarda ligacao no porto " + port+"...");  
                Socket newSock = serverSocket.accept(); // espera connect do cliente  
                // circuito virtual estabelecido: socket cliente <==> newSock  
                is = new BufferedReader(new InputStreamReader(newSock.getInputStream()));  
                os = new PrintWriter(newSock.getOutputStream(), true);  
                String inputLine = is.readLine();  
                System.out.println("Recebi -> " + inputLine);  
                os.println( "Olá para ti também!!");  
                //os.flush();  
                is.close();  
                os.close();  
                newSock.close();  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
            System.err.println("Erro na execução do servidor principal:"+e.getMessage());  
        }  
    }  
}
```

Cria o socket servidor
para esperar ligações

Espera
ligações

Criar Streams
para ler e
escrever do
socket

Recebe
mensagem do
socket

Envia
mensagem
para socket

Servidor Iterativo



Sockets UDP (1)

- ❏ Não existe diferenciação nos sockets clientes e servidores “DatagramSocket”
- ❏ Os sockets UDP enviam e recebem dados recorrendo a pacotes UDP, representados como objectos da classe “DatagramPacket”
 - ❏ Os DatagramSocket’s apenas fazem *send()* e *receive()* de DatagramPacket’s
- ❏ O socket é criado instanciando um objecto da classe “DatagramSocket” fornecendo no construtor o porto, caso este seja omitido é atribuído um porto automaticamente



Sockets UDP (2)

Criação dos Sockets (sem estabelecer ligação)

Cliente

Servidor

```
DatagramSocket udpSockSrv = new DatagramSocket(portoSrv)
```

```
DatagramSocket udpSockCli = new DatagramSocket()
```

```
String userInput = new String("Olá Mundo!!!");  
DatagramPacket p = new DatagramPacket(  
    userInput.getBytes(),  
    userInput.length(),  
    InetAddress.getByName("localhost"),  
    portoSrv);  
  
udpSockCli.send(p);
```

Envia Dados

**Fase de Transferência
de Dados**

```
byte[] buf = new byte[100];  
DatagramPacket rpacket = new DatagramPacket(buf,  
    buf.length);  
udpSockSrv.receive(rpacket);  
System.out.println("Dados recebidos: "  
    +new String(rpacket.getData()));
```

```
udpSockCli.close()
```




```
udpSockSrv.close()
```

Fecho dos sockets





Sockets UDP (3)

Criação do Socket udp (Classe java.net.DatagramSocket).

-  `public DatagramSocket() throws SocketException`
-  `public DatagramSocket(int port) throws SocketException`
-  `Etc...`

A transferência de dados é feita através de datagramas (Classe java.net.DatagramPacket)


-  `public void receive(DatagramPacket p) throws IOException`
-  `public void send(DatagramPacket p) throws IOException`

Fecho do socket, permite libertar thread's bloqueadas à espera de mensagens gerando SocketException.

-  `public void close()`



Sockets UDP – Datagramas (4)

 Os dados são enviados em pacotes representados pela classe `java.net.DatagramPacket`

 Criação de um pacote para recepção:

```
@ public DatagramPacket(byte[] buf, int length)
```

 Criação de um pacote para envio:

```
@ public DatagramPacket(byte[] buf, int length, InetAddress address,  
int port)
```

 Obter o endereço de origem / destino

```
@ public InetAddress getAddress()
```

 Obter o porto de origem / destino

```
@ public int getPort()
```



Exemplo – cliente UDP (3)

```
import java.io.*;
import java.net.*;
import java.util.*;
```

```
public class Client_udp {
    public static void main(String[] args) throws IOException {
        DatagramSocket sockfd = null;
        DatagramPacket epacket=null, rpacket=null;
        try { // Cria socket - UDP
            sockfd = new DatagramSocket();
        } catch (SocketException e) {
            e.printStackTrace();
            System.exit(-1);
        }
        // Mostrar os parametros da ligação
        System.out.println("Endereço do Servidor: " +
            InetAddress.getByNome(host) + " Porto: " + 5026);
        System.out.println("Endereço Local: "
            + sockfd.getLocalAddress() + " Porto: "
            + sockfd.getLocalPort());
        // constroi mensagem
        String userInput = new String("Olá mundo!!");
        // Envia pedido
        epacket =
            new DatagramPacket(
                userInput.getBytes(),
                userInput.length(),
                InetAddress.getByNome("localhost"), 5026);
```

Cria socket

Cria pacote UDP

```
    try {
        sockfd.send(epacket);
    } catch (IOException e) {
        e.printStackTrace();
        System.err.println("Erro no envio da mensagem: " + e.getMessage());
        System.exit(-1);
    }
    // Recebe resposta
    byte[] buf = new byte[100];
    rpacket = new DatagramPacket(buf, buf.length);
    try {
        sockfd.receive(rpacket);
    } catch (IOException e) {
        System.err.println("Erro na recepção da mensagem: " +
            e.getMessage());
        System.exit(-1);
    }
    // Mostra Resposta
    String received = new String(rpacket.getData(), 0,
        rpacket.getLength());
    System.out.println("Dados recebidos: " + received);
    // No fim de tudo fechar o socket
    sockfd.close();
}
```

Envia o pacote UDP

Recebe um pacote UDP



Exemplo – servidor UDP (4)

```
import java.net.*;
import java.io.*;
import sun.net.*;
```

```
class Servidor_udp {
public static void main(String args[]) {
    DatagramSocket sockfd = null;
    DatagramPacket epacket, rpacket;
    String host = "localhost";
    int dim_buffer = 100;
    byte ibuffer[] = new byte[dim_buffer];
    String received = null;

    try { // Cria socket - UDP
        sockfd = new DatagramSocket(5026);
    } catch (SocketException e) {
        e.printStackTrace();
        System.err.println("Erro na execução do servidor
(porto):" + e.getMessage());
        System.exit(-1);
    }
    rpacket = new DatagramPacket(ibuffer, dim_buffer);
    for (;;) {
        try {
            System.out.println( "Servidor aguarda recepção de
mensagem no porto 5026");
            rpacket.setLength(dim_buffer);
            sockfd.receive(rpacket);
        }
```

**Cria o socket servidor
para esperar
mensagens**

**Espera
mensagem**

```
        catch (IOException e) {
            e.printStackTrace();
            System.err.println("Erro na recepção da mensagem: " + e.getMessage());
            System.exit(-1);
        }
        received = new String(rpacket.getData(), 0, rpacket.getLength());
        // Criar um datagrama para enviar a resposta
        epacket = new DatagramPacket( received.getBytes(),
            received.length(), rpacket.getAddress(),
            rpacket.getPort());

        System.out.println("Endereço do cliente:" + epacket.getAddress() + " Porto:
" + epacket.getPort());
        System.out.println("Dados recebidos: " + received);
        System.out.println("Número de bytes recebidos: " +
            rpacket.getLength());

        try {
            sockfd.send(epacket);
        } catch (IOException e) {
            e.printStackTrace();
            System.err.println("Erro no envio da mensagem: " +
                e.getMessage());
            System.exit(-1);
        }
    } // Fim do ciclo do servidor
}
```

**Cria novo pacote para
enviar resposta**

**Envia
resposta**

Servidor Iterativo



Resumo: TCP *versus* UDP

- O TCP é usado em situações em que o cliente interage com o servidor de forma continuada, i.e., existe uma troca de informações durante um período significativo. Usa-se sempre que se necessita de entrega fiável das mensagens.
- O UDP deve ser forçosamente usado em aplicações que necessitem de fazer *broadcast* ou *multicast*.
- O UDP é frequentemente usado em aplicações do tipo em que existe um pedido e uma resposta, implementação simples: *timeout* e retransmissão.
- Uma aplicação que usa UDP pode necessitar de implementar algumas das facilidades disponibilizadas pelo TCP tipicamente:
 - Indicação de *acknowledge*
 - Controle de fluxo (*window flow control*)
 - Evitar congestionamento (*congestion avoidance*)
- O UDP não deve ser usado para transferência de blocos de dados (Ex: transferência de ficheiros), neste caso estaríamos a reinventar o TCP.
 - Exceções a esta regra são o TFTP e o NFS que são implementados em UDP por razões de eficiência.